

TECHNICAL REPORT

json2run: a tool for experiment design & analysis

Tommaso Urli *
tommaso.urli@uniud.it

May 7, 2013

Contents

| | | |
|----------|--|----------|
| 1 | Introduction | 2 |
| 2 | Installation | 3 |
| 3 | Usage | 4 |
| 3.1 | Designing experiment files | 4 |
| 3.1.1 | Basic JSON syntax | 4 |
| 3.1.2 | Combinations and alternatives (inner nodes) | 5 |
| 3.1.3 | Leaf nodes | 7 |
| 3.1.4 | Post-processors | 10 |
| 3.2 | The j2r command line tool | 14 |
| 3.2.1 | Input | 14 |
| 3.2.2 | Available actions | 14 |
| 3.2.3 | Batch options | 15 |
| 3.2.4 | Running options | 15 |
| 3.2.5 | Extra options for races | 16 |
| 3.2.6 | Database options | 16 |
| 3.2.7 | Logging info | 17 |
| 3.2.8 | Source code versioning | 17 |
| 3.2.9 | Instances and configurations | 17 |
| 3.3 | Running examples | 17 |
| 3.3.1 | Running a batch of experiments | 17 |
| 3.3.2 | Running a configuration race | 17 |
| 3.3.3 | Resuming a batch or a race | 18 |
| 3.3.4 | Printing detailed data about a batch or race | 18 |

* *Scheduling and Time-Tabling Group*, DIEGM - University of Udine, Via delle Scienze 206, 33100 – Udine (UD), Italy

| | | |
|-------|---|----|
| 3.3.5 | Print the list of winning (so far) configurations in a race | 18 |
| 3.3.6 | Delete a batch or a race from the database | 18 |
| 3.3.7 | List all the batches on the database | 18 |
| 3.4 | Analyzing the outcome | 18 |
| 4 | Future | 19 |
| 5 | Licensing | 19 |

1 Introduction

json2run is a tool to automate the running, storage and analysis of experiments. It has been created in the first place to study different algorithms or different sets of values for algorithm parameters, but it is a general tool and can be used wherever it fits. The main advantage of **json2run** (over a home-brewed experiment suite) is that it allows to describe a set of experiments concisely as a JSON-formatted parameter tree, such as the following (note the presence of parameter definitions as well as logical operators to combine them)

```
{
  "type": "and",
  "descendants": [
    {
      "type": "discrete",
      "name": "a",
      "values": [ "foo", "bar", "baz" ]
    },
    {
      "type": "or",
      "descendants": [
        {
          "type": "discrete",
          "name": "b1",
          "values": { "min": 0.0, "max": 1.0, "step": 0.25 }
        },
        {
          "type": "discrete",
          "name": "b2",
          "values": { "min": 2.0, "max": 10.0, "step": 2.0 }
        }
      ]
    }
  ]
}
```

An experiment file such as the one above describes the parameters that must be generated and passed over to the executable. We'll call a set of generated parameters a *configuration* or *parameter configuration*. Once the experiments have been described, **json2run** can parse the file and perform various operations with it, such as

- printing the generated configurations as command line options,

- running a batch of experiments based on the generated configurations,
- store the results of the experiments in a database,
- running a parameter race (see [1]) to find out the configurations (or configuration) that optimize a function of quality,
- retrieving the results of the experiments from the database.

In the first case, the outcome of our above example would be something like this (**json2run** comes in form of a command line tool called **j2r**):

```
$ j2r -i experiments.json
--a foo --b1 0.0
--a foo --b1 0.25
...
--a baz --b2 8.0
--a baz --b2 10.0
```

json2run supports a number of different types of nodes in the JSON tree, including: nodes for generating parameters from a discrete set of variables, nodes for generating parameters from the content of a directory or a file, nodes for sampling values from an interval, and so on. If something cannot be expressed with simple parameter generators, a number of **post-processors** allow you to mix, merge and discard the generated parameters in extremely flexible ways. Most post-processors were created because something couldn't be expressed with simple logical operators, but **json2run** slowly converged to something complete now.

The experiments results are stored on a (MongoDB) database, in order to be accessed later for analysis. The choice of MongoDB comes from the necessity of comparing algorithms which can have different (and a different number of) parameters, and a using a tabular storage (as in most relational database) would make queries much more difficult.

Finally, **json2run** comes with a very general but handy R script, which allows to gather data from the database, and do whatever kind of statistical analysis over it.

2 Installation

Being packaged as a python module, the installation of **json2run** should be quite straightforward, just ensure that the **bson** python module is not installed on your system (if this is the case, run `sudo pip uninstall bson`) since **pymongo** comes with its own **bson.*** classes and conflicts may occur. Then, clone the (Mercurial) repository and run the python installer

```
hg clone https://tunnuz@bitbucket.org/tunnuz/json2run
cd json2run
```

```
sudo python setup.py install
```

or, if you plan to update **json2run** often (i.e. if you plan to customize it, or update it from the repository), run

```
sudo python setup.py develop
```

this will allow you to update to the latest version by just running

```
hg pull -u
```

in the root directory.

3 Usage

Since **json2run** is designed to be very flexible (its only requirements being that you expose all the parameters of your executable and that you have access to a MongoDB instance), this also means that it comes with a lot of options. We will go through them in the following sections, but if you just need a quick reference type

```
$ j2r --help
```

Before running anything, however, you will need to know how to write an experiment file.

3.1 Designing experiment files

As previously mentioned, **json2run** expects a description of the experiments in JSON format. JSON (which, by the way, stands for JavaScript Object Notation) is a concise and human-readable language for describing structured data. It is also the very language MongoDB uses to store its data and to make queries, which makes for a natural integration.

3.1.1 Basic JSON syntax

The basic components of a JSON documents are arrays, objects and scalars. You can use array and object to group more arrays and objects, or scalars.

Scalars JSON has a number of native types for scalars:

- numbers,
- strings, and
- booleans.

Numbers can be integers or floats, and scientific notation is also supported.

Arrays and objects Arrays are lists of scalars separated by commas, e.g.:

```
[1, 2, 3, "foo", 3.14, true]
```

Objects can be seen as named arrays (similar to C++'s map, or dictionaries if you're into Python):

```
{
    "name": "John",
    "surname": "Boags",
    "profession": "beer maker",
    "age": 150
}
```

Note that arrays and objects use a different kind of parenthesis, {} vs [].

Comments JSON doesn't support comments, and most of the time you won't need them (the JSON contents should be sufficiently explanatory), but if you really need annotations, you can add fake entries to your objects, that won't be parsed by **json2run** and will serve as comments, such as **comment** in the following example:

```
{
    "type": "discrete",
    "name": "initial_temperature",
    "values": { "min": 10, "max": 30, "step": 10 }
    "comment": "The initial temperature for SA."
}
```

Specific syntax In particular, **json2run** assumes that the experiment file is a representation of a tree in which each node is a JSON object with at least a **type** field describing its type, e.g.:

```
{
    "type": "<node_type>",
    ...
}
```

We'll see the supported node types and their additional fields in the following sections.

3.1.2 Combinations and alternatives (inner nodes)

Typically, an algorithm accepts multiple parameters, and we want to be able to compare alternative combinations of these parameters. **and** and **or** nodes are the way to accomplish this in **json2run** and we call them *inner* nodes. Each inner node has a list of descendants and when they are activated, they either combine them together (in case of an **and**), or pick between them (in case of an **or**).

and nodes For instance, suppose that we have a Simulated Annealing [2] solver that accepts an initial temperature and a cooling schedule as command line parameters, and we would like to try all the possible combinations. In **json2run** this is expressed using an **and** node that combines the two parameters (ignore for now the syntax to describe discrete parameters, we'll come to that later).

```
{
  "type": "and",
  "descendants": [
    {
      "type": "discrete",
      "name": "initial_temperature",
      "values": { "min": 10, "max": 30, "step": 10 }
    },
    {
      "type": "discrete",
      "name": "cooling_schedule",
      "values": [ 0.999, 0.99, 0.9 ]
    }
  ]
}
```

```
$ j2r -i experiments.json
--initial_temperature 10.0 --cooling_schedule 0.999
--initial_temperature 10.0 --cooling_schedule 0.99
--initial_temperature 10.0 --cooling_schedule 0.9
--initial_temperature 20.0 --cooling_schedule 0.999
--initial_temperature 20.0 --cooling_schedule 0.99
--initial_temperature 20.0 --cooling_schedule 0.9
--initial_temperature 30.0 --cooling_schedule 0.999
--initial_temperature 30.0 --cooling_schedule 0.99
--initial_temperature 30.0 --cooling_schedule 0.9
```

or nodes Sometimes you have algorithms that accept different parameters and, possibly, a different number of them. Suppose your solver can operate either using Simulated Annealing and Tabu Search. You can easily encode this in an experiment file by using an **or** node.

```
{
  "type": "or",
  "descendants": [
    {
      "type": "and",
      "descendants": [
        {
          "type": "discrete",
          "name": "algorithm",
          "values": [ "sa" ]
        },
        {
          "type": "discrete",
          "name": "initial_temperature",
          "values": { "min": 10, "max": 20, "step": 10 }
        }
      ]
    }
  ]
}
```

```

    },
    {
      "type": "discrete",
      "name": "cooling_schedule",
      "values": [ 0.999, 0.99 ]
    }
  ]
},
{
  "type": "and",
  "descendants": [
    {
      "type": "discrete",
      "name": "algorithm",
      "values": [ "ts" ]
    },
    {
      "type": "discrete",
      "name": "tabu_list_length",
      "values": [ 10, 15, 20 ]
    }
  ]
}
]
}

$ j2r -i experiments.json
--algorithm sa --initial_temperature 10.0 --cooling_schedule 0.999
--algorithm sa --initial_temperature 10.0 --cooling_schedule 0.99
--algorithm sa --initial_temperature 20.0 --cooling_schedule 0.999
--algorithm sa --initial_temperature 20.0 --cooling_schedule 0.99
--algorithm ts --tabu_list_length 10
--algorithm ts --tabu_list_length 15
--algorithm ts --tabu_list_length 20

```

Note that no Tabu Search parameters appear in the Simulated Annealing configurations, and vice versa. By combining several **and** and **or** node, it is possible to express quite complex experiment designs.

3.1.3 Leaf nodes

Leaf nodes are responsible for creating values for single parameters. They all come with a **type**, a **name** for the parameter, and an array (or object) of **values** describing the possible values that the parameter can take, e.g.:

```

{
  "type": "<leaf_type>",
  "name": "<parameter_name>",
  "values": <values_definition>
}

```

discrete nodes Discrete nodes are the simplest kind of leaf nodes. They come with two value definition styles: an *explicit* one, where parameters are listed explicitly, e.g.:

```
{
  "type": "discrete",
  "name": "num_of_reheats",
  "values": [ 1, 2 ]
}
```

and an *implicit* one, which allows to define discrete numeric values from a `min`, a `max` and a `step`, e.g.:

```
{
  "type": "discrete",
  "name": "start_temperature",
  "values": { "min": 0.0, "max": 10.0, "step": 0.034 }
}
```

During the processing of the tree, implicit value definitions are transformed into explicit ones, and treated as such. A **discrete** node generates all the possible parameter values in order.

continuous nodes Continuous nodes allow to define parameter values that are generated by sampling continuous parameter spaces. These spaces are defined in terms of a `min` and a `max`, e.g.:

```
{
  "type": "continuous",
  "name": "start_temperature",
  "values": { "min": 0.0, "max": 10.0 }
}
```

However, continuous nodes can't generate parameter values by themselves. Instead, they need to be processed later on by a *post-processor* attached to a node upper in the tree hierarchy. This might seem over complicated, but there's a use case behind it.

In particular suppose that you want to study the interaction of two parameters on the performance of an algorithm. To do a proper sampling, the generated parameters must be picked from a 2-dimensional space, in a way that is as uniform as possible. One way to do this, would be to generate an **and** node containing several **discrete** nodes with different ranges. There are two problems with this approach (which is called *full-factorial*):

1. the generated points are very regular, while one usually want to sample the parameter space randomly (but uniformly),
2. the parameter combinations are the cartesian product of the values generated for each single parameter, which makes it difficult to control how much configurations are generated.

While this can still be done with **json2run** (and indeed is often what one wants), we would like to treat the parameter space generated by the interaction of the two parameters as a *single* space, and sample 2-dimensional points uniformly inside it. **json2run** comes with a post-processor which is able to generate the Hammersley point set in a k-dimensional space. We will see in the section about post-processors how to attach one to a node, but for the moment just accept that **continuous** leaf nodes are treated in this special way.

file and directory nodes File (or directory) nodes are essentially **discrete** nodes, whose values are nor defined explicitly nor implicitly, but instead are generated from the content of a file (or a directory). The typical use for this kind of nodes is to generate experiments that run on a set of instances specified in a file e.g.:

```
{
  "type": "file",
  "name": "instance",
  "path": "selected_instances.txt",
  "match": ".*"
}
```

where the **path** field specifies the location of the file to be used as input, and the **match** field restrict the generated parameters to the lines of the file matching a given regular expression¹. This is useful when you want to restrict to certain instances, but most frequently will just be “.” (catch-all). As for directory nodes, they follow a similar semantic, the difference being that the generated values is the list of the content of the directory, filtered by the regular expression in the **match** field.

```
{
  "type": "directory",
  "path": "../instances/comp",
  "name": "instance",
  "match": ".*\\.ectt"
}
```

flag nodes Flag nodes have a single parameter (the **name** of the generated flag) and generate value-less parameters. E.g.:

```
{
  "type": "flag",
  "name": "verbose"
}
```

¹**Note** regular expressions are in Python format, but strings must be escaped, e.g. if you want to look for the .txt pattern, the string must be specified as “.*\\.txt”

will just add the `--verbose` flag on the generated command lines.

3.1.4 Post-processors

Post-processors are tools to generate more complex combinations of parameters. They can be attached to **any** inner node by adding them to a field called `postprocessors` along with the descendants, e.g.:

```
{
  "type": "and",
  "descendants": [
    ...
  ],
  "postprocessors": [
    ...
  ]
}
```

Post-processors have a `type` field and a number of other fields dependent on the specific post-processor type. They all operate in the following way:

1. They take the list of parameters generated in the subtree (note that each execution of a subtree gives birth to a different parameter configuration) they are attached to,
2. they process the list **as a whole** (e.g. replacing parameters, modifying values, removing parameters, and so forth), and finally
3. they return the new list, which replaces the old one.

In many cases they just apply the same function over all the elements of the list, but they might be designed to do more complex things or to just update certain kind of parameters. For instance, the `hammersley` post-processor only apply to `continuous` parameters (but possibly more than one of them at a time).

Note order matters! Post-processors are applied in the order in which they are defined in the `postprocessors` list.

expression processors Expression post processors are by far the most flexible ones. They allow to define a new parameter (either `discrete` or `continuous`, but **not** a flag) by evaluating a python expression and using the result as the value of the parameter. The processor is defined by a `match` regular expression, which captures the operands needed by the expression, and either

1. an `expression` which will be evaluated to yield the value of the generated `discrete` parameter, or

2. two expressions (`min` and `max`) that will yield the values for the minimum and maximum of the generated `continuous` parameter.

The type of the parameter is inferred by the presence of the `expression` field, while its name is defined by the `result` field. An example of the two syntaxes is the following:

```
{
  "type": "expression",
  "match": "<operand_1>|<operand_2>|...",
  "result": "<parameter_name>",
  "expression": "<expression>"
}

{
  "type": "expression",
  "match": "<operand_1>|<operand_2>|...",
  "result": "<parameter_name>",
  "min": "<min_expression>",
  "max": "<max_expression>"
}
```

Expression syntax Any valid python expression that has a return value can be used as `expression`, `min` or `max`. To access the values of the captured operands it is sufficient to postfix their name with `.value`.

As for the available operations, functions from python's `math` and `json` modules are automatically imported. For instance, to generate a new parameter `p3` which is the power of two existing ones, `p1` and `p2`, we'll write:

```
{
  "type": "expression",
  "match": "p1|p2",
  "result": "p3",
  "expression": "pow(p1.value, p2.value)"
}
```

While to generate a parameter which takes values in $[0.1*p1, 5*p2]$, we'll do:

```
{
  "type": "expression",
  "match": "p1|p2",
  "result": "p3",
  "min": "0.1*p1.value",
  "max": "5*p2.value"
}
```

ignore processors Ignore post processors can be used to remove specific parameters from the list of generated ones. Typically, they are used to discard operands of an **expression** post-processor after they have been used. Following the previous example, we might not be interested in *p1* and *p2* at all, so:

```
{
  "type": "ignore",
  "match": "p1|p2"
}
```

Remember that post-processors are applied in order, thus (in this case) the **ignore** must be defined after the **expression**.

sorting processors Sorting allows to define an ordering for a subset of parameters. These parameters will be put (if they exist) at the beginning of the generated list of parameters, and the others will follow. The syntax of the post-processor is the following:

```
{
  "type": "sorting",
  "order": [ "<param_1>", "<param_2>", ... ]
}
```

Where **order** is an array of ordered parameter names.

hammersley processors The Hammersley post-processor generates the scaled k-dimensional Hammersley point set from a set of k **continuous** parameters, and it's the preferential (also, the only) way to sample continuous parameter spaces. The syntax is the following:

```
{
  "type": "hammersley",
  "points": <n>
}
```

So, assuming that your experiments file produces k continuous parameters, the `hammersley` post-processor generates a k -dimensional *cube* delimited by the `min` and `max` fields of your `continuous` parameters, and will generate `n` samples inside this cube, to use as parameter values.

The Hammersley point set This choice of the Hammersley point set has been driven by two properties of this sequence that make it favourable to parameter tuning. First, points from the Hammersley set exhibit *low discrepancy*, i.e. they are well distributed across the parameter space despite being random-like. Second, the sequence is *scalable* both with respect to the number of points (`n`) to be sampled and to the number of dimensions (`k`) of the sampling space.

So, whenever you want to explore parameter spaces, use `continuous` parameters and the `hammersley` post-processor.

rounding processors When sampling continuous parameters or using `expression` post-processors, the resulting values can end up being floats with many decimal digits. While this in general is not an issue, often this much precision is unneeded, and it's just more convenient to operate with less precise floats. The `rounding` post-processor allows to round down a parameter's values to a specific number of decimal digits. The syntax is the following:

```
{
  "type": "rounding",
  "match": "<regex>"
  "decimal_digits": <n>
}
```

Where `n` is the number of decimal digits we want to retain (**note** the numbers are rounded, not truncated, to `n` digits after the floating point), and `match` is a regular expression describing the parameters we want to round down.

Compact syntax Rounding post-processors also support a compact syntax, to group roundings in a single post-processor. The syntax is the following:

```
{
  "type": "round",
  "round": [
    "<regex_1>": <n_1>,
    "<regex_2>": <n_2>,
    ...
  ]
}
```

Where `regex_k` are regular expressions describing one (or more) parameter, and `n_k` are the corresponding decimal digits we want to retain.

Forcing precision Both syntaxes support an optional field called `force_precision`, which can be either `true` or `false`, that forces the resulting value to have the specified number of decimal digits (regardless of any rounding to zero).

renaming processors Rename post-processors can be used to rename parameters. The syntax, somewhat similar to `rounding`'s compact one, is the following:

```
{
  "type": "renaming",
  "rename": {
    "old_1": "new_1",
    "old_2": "new_2",
    ...
  }
}
```

Where `old_k` are the original name of the parameters we want to rename and `new_k` are the new ones. Note that unlike `rounding`'s compact syntax, here `rename` is an object, not an array. Also, `old_k` are plain strings, not regular expressions.

3.2 The j2r command line tool

All of **json2run** functionalities are accessed through a command line utility called `j2r`. The tool comes with a (large) number of options, activated by `--` followed by their long names, or equivalently `-` followed by their short names. Some of them have default values, some other must be provided. (See `j2r --help` for a summary of them.)

3.2.1 Input

Most of the functionalities of **json2run** require that you provide an input file, i.e. a JSON file describing experiments. This file is specified through the `--input` (or `-i`) option.

```
$ j2r -i experiments.json
```

3.2.2 Available actions

The main switch in `j2r` is the `--action` (or `-a`) option. Actions allow to specify what you want **json2run** to do for you. The available actions are:

- **print-cll** prints the generated experiments as a *command line list* (also, the default option)
- **print-csv** print the generated experiments as a CSV file
- **run-batch** start (or resumes) a batch for the experiments described in the input file
- **run-race** starts (or resumes) a race among parameter configurations in order to find the best parameter assignment for the specified executable
- **list-batches** list the batches on the database, and give summary information about their completion, machine on which they are being run, type of batch, etc.
- **delete-batch** delete a batch from the database
- **batch-info** provides detailed information about a batch or race, e.g. repetitions completed, configurations that are still racing, experiment file, etc.
- **rename-batch** rename a batch on the database
- **show-winning** show winning configurations in a race
- **set-repetitions** set the number of repetitions of the same experiments in a batch or a race
- **dump-experiments** dump all the experiments data regarding a batch as a CSV file
- **mark-unfinished** set a batch as unfinished, in order to restart it

3.2.3 Batch options

Some of the actions require additional parameters, in particular, each action pertaining a batch on the database must also provide a mandatory **--batch-name** (or **-n**) to refer it (see it as a key for batches in the database).

3.2.4 Running options

Both **run-batch** and **run-race** have a number of additional parameters that tune the way in which the experiments are run.

- **--executable** (or **-e**) specifies the executable to be run (mandatory)
- **--parallel-threads** (or **-p**) specifies the maximum number of parallel processors to run the experiments onto (defaults to the number of cores on the machine where the experiments are run)

- `--repetitions` (or `-r`) number of repetitions of the (exactly) same experiment to run (e.g. to have a more reliable result)
- `--greedy` (or `-g`) can be `true` or `false` and states if the batch or race can reuse experiments which are already on the database (but are possibly part of other batches and races)

3.2.5 Extra options for races

When a race is run, a number of additional parameters must or can be passed.

- `--instance-param` (or `-ip`) specifies the parameter which represents the instance
- `--performance-param` (or `-pp`) specifies the statistic (output by the executable), that must be used to evaluate the quality of a configuration
- `--seed` (or `-s`) seed to use to shuffle instances (defaults to 0)
- `--confidence` confidence level for hypothesis testing (e.g. to compare with p-values, defaults to *0.05*)

Note `json2run` assumes that the executable outputs a valid JSON code, with a field for each statistic that we want to record. For instance, a solver could have a `cost` and a `time` statistic.

```
{
  "cost": 161.12,
  "time": 500
}
```

3.2.6 Database options

When we're running batches or databases, we're implicitly assuming that we have a running and accessible MongoDB database. By default, `json2run` will look for MongoDB on the `localhost` and will try to connect to the database `j2r` with username `j2r` and password `j2r`. These are just convenient credentials, but one can specify its own connection parameters through the following options.

- `--db-host` (or `-dh`) specifies the host onto which the MongoDB instance is running
- `--db-database` (or `-dd`) specifies the database to use connecting

- `--db-user` (or `-du`) specifies the username to use for connecting
- `--db-pass` (or `-dx`) specifies the password to use for connecting
- `--db-port` (or `-dp`) specifies the port to use for connecting

3.2.7 Logging info

By default `j2r` prints on the standard output most of its logging information. However this information can be redirected on a file if needed, and the log level can be set.

- `--log-file` specifies the file where the log is written (default: None)
- `--log-level` can be `warning`, `error`, `info`

3.2.8 Source code versioning

Additionally, `json2run` can record the code revision used for running a batch or a race. To enable this option one must pass the name of the source code manager of choice through the `--scm` option (currently supports `git` and `mercurial`).

3.2.9 Instances and configurations

Instances and parameter configurations are described in the same experiments file.

3.3 Running examples

Here are some of the most common operations that one can perform with `json2run`.

3.3.1 Running a batch of experiments

Run a batch of experiments based on an experiment file (*experiments.json*) and an executable (*solver*), with 10 repetitions for each experiment and all the available cores.

```
$ j2r -a run-batch -r 10 -n my_batch -i experiments.json -e ./solver
```

3.3.2 Running a configuration race

Based on the same file, and reckoning that the instance parameter is called *instance*, we can run a race to find out the best configuration. Suppose that the solver outputs some statistics in JSON (as in the example above) and that we want to compare the configurations based on the *cost* of the obtained solutions.

```
$ j2r -a run-race -r 10 -n my_race -i experiments.json -ip instance -  
pp cost -e ./solver
```

3.3.3 Resuming a batch or a race

To resume a previously stopped batch or race, it is sufficient to run

```
$ j2r -a run-batch -n my_batch
```

or

```
$ j2r -a run-race -n my_race
```

3.3.4 Printing detailed data about a batch or race

Use the `batch-info` action, passing the name of the race or batch.

```
$ j2r -a batch-info -n my_race
```

the output is in JSON format (for easy parsing by other tools).

3.3.5 Print the list of winning (so far) configurations in a race

Use the `show-winning` action, passing the name of the race or batch.

```
$ j2r -a show-winning -n my_race
```

3.3.6 Delete a batch or a race from the database

Use the `delete-batch` action, passing the name of the race or batch.

```
$ j2r -a delete-batch -n my_race
```

3.3.7 List all the batches on the database

Use the `list-batches` action.

```
$ j2r -a list-batches
```

3.4 Analyzing the outcome

The outcome of a batch or race, i.e. all the data about the experiments, can be retrieved from R by loading the R script `analysis.R` and using the following functions:

```
source("analysis.R")  
connect("host")
```

```
x <- getExperiments("my_race", c("instance"))
```

The `x` data frame will contain a row for each experiment in the batch or race, with information about whether the configuration was one of the winning ones (in case of a race).

4 Future

A new major version of **json2run** is in the works. Among the upcoming features are:

- launching of experiments on multiple machines,
- web-service, i.e. RESTful, infrastructure will handle all **json2run** operations,
- improved JSON syntax for all node types (fields and type of values will determine which kind of node are we dealing with), e.g.:

```
{
  "type": "and",
  "descendants": [ ... ]
}
```

will become:

```
{
  "and": [ ... ]
}
```

and

```
{
  "type": "discrete",
  "name": "param1"
  "values": { "min": 0.1, "max": 1.0, "step": 0.1 }
}
```

will become:

```
{
  "param1": { "min": 0.1, "max": 1.0, "step": 0.1 }
}
```

5 Licensing

json2run is open-source and distributed under the MIT license.

Acknowledgements

json2run has been developed for and with the collaboration of Luca Di Gaspero, Sara Ceschia and Andrea Schaerf of the Scheduling and Time-Tabling Group of University of Udine. Thanks go to Tiago Januario from Universidade Federal de Minas Gerais, Belo Horizonte, Brasil for the many suggestions. Also, thanks go to the Bitbucket staff that hosts **json2run**'s code free of charge.

References

- [1] Mauro Birattari, Zhi Yuan, and Thomas Stützle. F-Race and iterated F-Race : An overview. . . . *methods for the analysis . . .*, (June), 2010.
- [2] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.